Clase 5 — 12.12.25

#VSC #javascript

👩‍🏫 Profesora: Sara Gonzalo
📘 Desarrollo de interfaces. JAVASCRIPT, JQUERY, REALIDAD VIRTUAL
🗓️ Clase 5 — 12/12/2025
🎯 **Tema:** Funciones (tipos y parámetros) + Eventos (HTML vs addEventListener) + Primer contacto con DOM

1️⃣ ¿Qué es una función y para qué sirve?

- Una **función** sirve para **agrupar código** y **reutilizarlo** cuando se necesite.
- En JavaScript, al **gestionar el DOM**, una acción sobre un elemento (nodo) puede **desencadenar** la ejecución de una función.

📌 Esquema de clase:
COMPORTAMIENTO = ACCIÓN (EVENTO) + FUNCIÓN

Ejemplo mental:

- Acción/evento: “pulsar un botón”
- Función: “mostrar un mensaje (alert)”

2️⃣ Funciones en JavaScript

🔹 2.1 ¿Qué es una función?

Una **función** es un **bloque de código reutilizable** que agrupa una serie de instrucciones y que:

- **no se ejecuta automáticamente**
- se ejecuta **solo cuando es llamada**
- permite organizar y reutilizar código
- se utiliza para responder a eventos (click, load, etc.)

📌 En JavaScript, cuando ocurre una **acción sobre un elemento HTML**, se suele **ejecutar una función**.

> Acción (evento) → ejecución de una función

🔹 2.2 ¿Por qué usar funciones?

Sin funciones, el código:

- se repite
- es difícil de mantener
- crece de forma desordenada

Con funciones:

- el código es más claro
- se reutiliza fácilmente
- los cambios se hacen en un solo sitio

🧪 Ejemplo sin función

```
window.alert("Hola");
window.alert("Hola");
window.alert("Hola");
```

🧪 Ejemplo con función

```
function mostrarMensaje() {
  window.alert("Hola");
}

mostrarMensaje();
mostrarMensaje();
mostrarMensaje();
```

✅ Mismo resultado
✅ Código más limpio y reutilizable

🔹 2.3 Forma básica de definir una función (procedimiento convencional)

📌 Sintaxis general

```
function nombreFuncion() {
  // instrucciones a ejecutar
}
```

🔍 Elementos de la definición

- `function` → palabra reservada
- `nombreFuncion` → nombre de la función
- `{}` → bloque de código
- El contenido **no se ejecuta al definirse**

🧪 Ejemplo práctico

```
function saludar() {
  window.alert("Buenas tardes");
}
```

📌 En este punto:

- la función está definida
- **todavía no se ejecuta**

🔹 2.4 Llamada (ejecución) de una función

Para que una función se ejecute, hay que **invocarla** explícitamente.

📌 Sintaxis de llamada

```
saludar();
```

🧪 Ejemplo completo

```
function saludar() {
  window.alert("Buenas tardes");
}

saludar(); // llamada a la función
```

📌 Importante:

- Los paréntesis `()` son obligatorios
- Sin llamada, la función no hace nada

🔹 2.5 Funciones que escriben en el documento (`document.write`)

Ejemplo que aparece en los apuntes:

```
function saludar() {
  document.write("Buenas tardes!!!");
}

saludar();
```

⚠️ **Observación importante**:

- `document.write()` escribe directamente en el HTML
- Si se ejecuta tras la carga de la página, puede sobrescribir el contenido
- Se usa principalmente con fines **didácticos**

🔹 2.6 Funciones almacenadas en variables

(forma alternativa de definición)

En JavaScript, una función también puede guardarse en una variable.

📌 Sintaxis

```
let nombreFuncion = function () {
  // instrucciones
};
```

🧪 Ejemplo equivalente al anterior

```
let saludar = function () {
  window.alert("Buenas tardes");
};

saludar();
```

🧠 Idea clave

- La función se trata como un valor
- Es muy habitual en:
  - eventos
  - listeners
  - callbacks

📌 **No es un tipo de función distinto**, solo **otra forma de definirla**.

🔹 2.7 Relación inicial entre funciones y eventos

Las funciones suelen ejecutarse como consecuencia de una acción:

```
function mostrarVentana() {
  window.alert("Evento ejecutado");
}
```

Más adelante se verá:

```
boton.addEventListener("click", mostrarVentana);
```

📌 Por ahora, quédate con esta idea:

> **Un evento ejecuta una función**

3️⃣ Parámetros de entrada en las funciones

🔹 3.1 ¿Qué son los parámetros de entrada?

Los **parámetros de entrada** son **variables** que se definen en la función y que reciben valores cuando la función es llamada.

📌 Permiten que una función:

- no trabaje siempre con los mismos datos
- se adapte a diferentes situaciones
- sea reutilizable

🔹 3.2 Definición de parámetros en una función

Los parámetros se definen **entre los paréntesis** de la función.

📌 Sintaxis general

```
function nombreFuncion(param1, param2) {
  // instrucciones
}
```

- `param1`, `param2` → parámetros (variables locales)
- Existen **solo dentro de la función**

🧪 Ejemplo básico

```
function mostrarMensaje(mensaje) {
  window.alert(mensaje);
}
```

📌 La función está preparada para recibir **un dato externo**.

🔹 3.3 Llamada a una función con parámetros (argumentos)

Cuando se llama a la función, se pasan los **argumentos**, que son los valores reales.

```
mostrarMensaje("Hola");
mostrarMensaje("Buenas tardes");
```

📌 Diferencia clave:

- **Parámetro** → variable en la definición
- **Argumento** → valor que se envía al llamar

🔹 3.4 Ejemplo completo comentado

```
function saludar(nombre) {
  window.alert("Hola " + nombre);
}

saludar("Sara");
saludar("Juan");
saludar("David");
```

🧠 Qué ocurre paso a paso

1. Se define la función `saludar`
2. `nombre` es un parámetro
3. Cada llamada envía un argumento distinto
4. La misma función produce resultados diferentes

🔹 3.5 Funciones con varios parámetros

Una función puede recibir **más de un parámetro**.

📌 Ejemplo

```
function sumar(a, b) {
  let resultado = a + b;
  window.alert("La suma es: " + resultado);
}

sumar(5, 3);
sumar(10, 20);
```

📌 Importante:

- El orden de los argumentos **importa**
- `a` recibe el primer valor, `b` el segundo

🔹 3.6 Parámetros como variables locales

Los parámetros:

- se comportan como variables
- **solo existen dentro de la función**
- no pueden usarse fuera

🧪 Ejemplo

```
function ejemplo(valor) {
  let total = valor * 2;
  console.log(total);
}

ejemplo(5);

// console.log(valor); // ❌ Error: no existe fuera
```

📌 Esto enlaza con el concepto de **ámbito (scope)**, que se verá con variables globales y locales.

🔹 3.7 Parámetros y eventos (primer contacto)

Los parámetros se usan mucho con eventos, por ejemplo:

```
function mostrarTexto(texto) {
  window.alert(texto);
}
```

Más adelante:

```
boton.addEventListener("click", function () {
  mostrarTexto("Has hecho click en el botón");
});
```

📌 El evento **no ejecuta directamente el código**, sino que:

> ejecuta una función **pasándole información**

🔹 3.8 Errores comunes con parámetros

❌ Llamar sin pasar valores:

```
function saludar(nombre) {
  alert(nombre);
}

saludar(); // nombre = undefined
```

❌ Confundir parámetros con argumentos
❌ Usar parámetros fuera de la función

4️⃣ Parámetros de salida (`return`) y tipos de funciones

🔹 4.1 Parámetros de salida: la instrucción `return`

Una función puede **devolver un valor** al punto donde ha sido llamada.

📌 Esto se hace con la palabra reservada `return` .

📌 Sintaxis general

```
function nombreFuncion() {
  return valor;
}
```

🧪 Ejemplo básico

```
function calcularSuma() {
  let suma = 23 + 45;
  return suma;
}
```

📌 La función **no muestra nada**, solo devuelve un valor.

🔹 4.2 Recoger el valor devuelto por una función

El valor que devuelve una función se puede:

- guardar en una variable
- usar en una expresión
- mostrar por pantalla

🧪 Ejemplo

```
let resultado = calcularSuma();
window.alert(resultado);
```

📌 Flujo:

1. Se ejecuta la función
2. `return` envía el valor
3. Se guarda en `resultado`
4. Se usa fuera de la función

🔹 4.3 Efecto de `return` dentro de una función

📌 **Muy importante (nivel examen)**
Cuando se ejecuta un `return`:

- la función **termina inmediatamente**
- el código posterior **no se ejecuta**

🧪 Ejemplo

```
function ejemploReturn() {
  return 10;
  console.log("Esto no se ejecuta");
}
```

🔹 4.4 Diferencia entre `return` y mostrar datos (`alert`, `document.write`)

`return`	`alert` / `document.write`
Devuelve un valor	Muestra información
Lógica interna	Salida visual
Reutilizable	No reutilizable
Profesional	Didáctico

📌 Una buena función **calcula**, no decide cómo mostrar el resultado.

🔹 4.5 Tipos de funciones (según entrada y salida)

> 📌 **Esta clasificación es la que aparece en el bloc de notas de clase**

Se clasifican según:

- si reciben parámetros
- si devuelven un valor

🔹 4.6 Tipo 1 — Funciones SIN parámetros y CON retorno

📌 Características

- no reciben datos externos
- realizan un cálculo interno
- devuelven un valor

🧪 Ejemplo

```
function calcular() {
  let suma = 23 + 45;
  return suma;
}

let resultado = calcular();
```

📌 Uso típico:

- valores fijos
- pruebas
- ejemplos didácticos

🔹 4.7 Tipo 2 — Funciones SIN parámetros y SIN retorno

📌 Características

- no reciben datos
- no devuelven nada
- solo ejecutan una acción

🧪 Ejemplo

```
function imprimir() {
  window.alert("Buenas tardes");
}

imprimir();
```

📌 Uso típico:

- mostrar mensajes
- lanzar alertas
- ejecutar acciones simples

🔹 4.8 Tipo 3 — Funciones CON parámetros y CON retorno

📌 Características

- reciben datos
- procesan información
- devuelven un resultado

🧪 Ejemplo (bloc de notas)

```
function calcularOperacion(p1, p2, p3) {
  let suma = p1 + p2 + p3;
  return suma;
}

let resultado = calcularOperacion(12, 24, 56);
```

📌 Es el tipo **más potente y reutilizable**.

🔹 4.9 Tipo 4 — Funciones CON parámetros y SIN retorno

📌 Características

- reciben datos
- no devuelven valores
- muestran o ejecutan acciones

🧪 Ejemplo

```
function calcular22(p1, p2) {
  let op = p1 + p2;
  window.alert("El resultado es: " + op);
}

calcular22(10, 5);
```

🔹 4.10 Esquema resumen de los tipos de funciones

```
                ¿Recibe parámetros?
                  /            \
               NO               SÍ
              /                   \
¿Devuelve valor?            ¿Devuelve valor?
     /      \                    /      \
   SÍ       NO                 SÍ       NO
```

🔹 4.11 Buenas prácticas con `return`

✅ Usar `return` para lógica
❌ No mezclar lógica y presentación
✅ Nombrar funciones según lo que devuelven
❌ Usar `return` solo para “mostrar cosas”

5️⃣ Variables globales y locales (ámbito o *scope*)

🔹 5.1 ¿Qué es el ámbito (*scope*) de una variable?

El **ámbito** (*scope*) indica:

- **dónde se crea** una variable
- **desde dónde puede usarse**
- **cuándo deja de existir**

📌 En JavaScript, el ámbito depende de:

- dónde se declara la variable
- si está dentro o fuera de una función

🔹 5.2 Variables globales

📌 Definición

Una **variable global** es aquella que:

- se declara **fuera de cualquier función**
- es accesible desde **todo el programa**

🧪 Ejemplo

```
var contador = 0; // variable global

function aumentar() {
  contador++;
  console.log(contador);
}

aumentar();
aumentar();
```

📌 `contador`:

- se declara fuera
- puede usarse dentro de la función
- mantiene su valor entre llamadas

⚠️ Observación importante (nivel examen)

- Las variables globales:
  - ocupan memoria durante toda la ejecución
  - pueden modificarse desde cualquier parte
  - pueden provocar errores difíciles de detectar

👉 Se deben usar **con cuidado**.

🔹 5.3 Variables locales

📌 Definición

Una **variable local** es aquella que:

- se declara **dentro de una función**
- solo existe mientras la función se ejecuta
- **no es accesible desde fuera**

🧪 Ejemplo

```
function calcular() {
  let resultado = 10 + 5; // variable local
  console.log(resultado);
}
```

```
calcular();

// console.log(resultado); // ❌ Error: no existe fuera
```

📌 `resultado`:

- nace dentro de la función
- muere al terminar la función

🔹 5.4 Diferencia práctica entre variable global y local

🧪 Ejemplo comparativo

```
var total = 0; // global

function sumar(valor) {
  let parcial = valor + 5; // local
  total += parcial;
  console.log("Total:", total);
}

sumar(5);
sumar(10);
```

🧠 Análisis

- `total`:
  - se conserva entre llamadas
  - es global
- `parcial`:
  - se crea y destruye en cada llamada
  - es local

🔹 5.5 Uso de `var` y `let` (según lo visto en clase)

En los apuntes aparece:

- `var` → variable global
- `let` → variable local

📌 Ejemplo típico de clase

```
var global = 10;

function ejemplo() {
  let local = 5;
  console.log(global + local);
}

ejemplo();
```

📌 Idea clave:

> `let` limita el ámbito
> `var` lo amplía

(Más adelante se verá que `var` tiene más implicaciones, pero de momento esta distinción es suficiente para clase.)

🔹 5.6 Variables locales con el mismo nombre

Una variable local puede tener el **mismo nombre** que una global.

🧪 Ejemplo

```
var numero = 10;

function prueba() {
  let numero = 5; // local
  console.log(numero);
}

prueba();
console.log(numero);
```

🧠 Resultado

- Dentro de la función → `5`
- Fuera de la función → `10`

📌 La variable local **tiene prioridad** dentro de su ámbito.

🔹 5.7 Variables y eventos (relación directa)

Cuando trabajemos con eventos:

- las funciones asociadas a eventos
- crean su propio ámbito
- usan variables locales

🧪 Ejemplo

```
var contador = 0;

boton.addEventListener("click", function () {
  let mensaje = "Has hecho click";
  contador++;
  console.log(mensaje, contador);
});
```

📌 `mensaje`:

- solo existe durante el evento
  📌 `contador`:
- conserva su valor entre clicks

🔹 5.8 Errores comunes con variables

❌ Usar variables locales fuera de su función
❌ Abusar de variables globales
❌ Reutilizar nombres sin entender el ámbito

🔹 5.9 Buenas prácticas

✅ Usar variables locales siempre que sea posible
✅ Usar globales solo cuando sea necesario
✅ Nombres claros y coherentes
❌ Evitar depender de globales en exceso

6️⃣ Eventos en JavaScript

🔹 6.1 ¿Qué es un evento?

Un **evento** es una **acción** que ocurre en el navegador y que puede ser **detectada por JavaScript**.

📌 Un evento puede producirse por:

- el usuario (click, teclado, ratón…)
- el navegador (carga de la página, cambio de tamaño…)

👉 Cuando ocurre un evento, **se ejecuta una función**.

🔹 6.2 Ejemplos de eventos cotidianos

Acción	Evento
Pulsar un botón	`click`
Pulsar una tecla	`keydown`
Soltar una tecla	`keyup`
Cargar la página	`load`
Enviar un formulario	`submit`
Pasar el ratón	`mousemove`

📌 Todos estos eventos pueden capturarse con JavaScript.

🔹 6.3 Modelo mental: evento → función

```
Usuario / Navegador
        ↓
     Evento
        ↓
    Función JS
```

Ejemplo conceptual:

```
function mostrarMensaje() {
  window.alert("Evento detectado");
}
```

👉 Esta función **no hace nada** hasta que un evento la ejecute.

🔹 6.4 Eventos y funciones: relación directa

Los eventos **no contienen código**, solo **lanzan funciones**.

🧪 Ejemplo

```
function saludar() {
  window.alert("Hola");
}
```

Más adelante:

```
boton.addEventListener("click", saludar);
```

📌 El evento `click` **dispara** la función `saludar`.

🔹 6.5 ¿Dónde se pueden aplicar eventos?

Los eventos pueden asociarse a:

- elementos HTML (`button` , `input` , `form` , etc.)
- el documento (`document`)
- la ventana del navegador (`window`)

🧪 Ejemplos

```
boton.addEventListener("click", funcion);
document.addEventListener("keydown", funcion);
window.addEventListener("load", funcion);
```

🔹 6.6 Eventos del ratón (introducción)

Algunos eventos de ratón:

- `click`
- `dblclick`
- `mousedown`
- `mouseup`
- `mousemove`

🧪 Ejemplo básico

```
boton.addEventListener("click", function () {
  console.log("Click detectado");
});
```

📌 El código dentro se ejecuta **solo cuando ocurre el evento**.

🔹 6.7 Eventos del teclado (introducción)

Eventos de teclado:

- `keydown`
- `keyup`

🧪 Ejemplo

```
document.addEventListener("keydown", function () {
  console.log("Tecla pulsada");
});
```

📌 El evento se asocia normalmente al `document`.

🔹 6.8 Eventos automáticos del navegador

Eventos que **no dependen del usuario**:

- `load`
- `DOMContentLoaded`

🧪 Ejemplo

```
window.addEventListener("load", function () {
  console.log("Página cargada completamente");
});
```

📌 Se usan para saber **cuándo está lista la página**.

🔹 6.9 Errores comunes con eventos

❌ Pensar que el evento contiene el código
❌ Ejecutar la función en vez de pasarla

```
// ❌ MAL
boton.addEventListener("click", saludar());

// ✅ BIEN
boton.addEventListener("click", saludar);
```

7️⃣ Gestión de eventos

🔹 7.1 Eventos definidos directamente en HTML

📌 Idea clave

El evento se escribe **dentro del propio HTML**, y desde ahí se llama a una función JavaScript.

🧪 Ejemplo típico

```
<body onload="mostrarVentana()">
  <button onclick="mostrarVentana()">Botón</button>

  <script src="app.js"></script>
</body>
```

```
function mostrarVentana() {
  window.alert("Evento lanzado desde HTML");
}
```

🧠 Qué está ocurriendo

- El HTML:
  - decide **cuándo** se ejecuta la función
  - contiene lógica de comportamiento
- JavaScript:
  - solo define la función

📌 El HTML **manda** sobre el comportamiento.

❌ Desventajas de este enfoque

- Mezcla estructura (HTML) y comportamiento (JS)
- HTML más sucio y difícil de mantener
- Poco escalable
- Difícil reutilización del código

👉 Se usa solo en ejemplos básicos o introducción.

🔹 7.2 Eventos gestionados desde JavaScript (`addEventListener`)

📌 Idea clave

El HTML **no tiene eventos**.
JavaScript:

- selecciona el elemento
- escucha el evento
- ejecuta la función

🧪 HTML limpio

```
<body>
  <button id="btn">Botón</button>
  <script src="app.js"></script>
</body>
```

🧪 JavaScript con listener

```
let boton = document.getElementById("btn");

boton.addEventListener("click", function () {
  window.alert("Evento gestionado desde JavaScript");
});
```

🧠 Qué está ocurriendo

1. El HTML define solo estructura
2. JS selecciona el elemento
3. JS escucha el evento (`click`)
4. JS ejecuta la función

📌 Aquí **JavaScript controla el comportamiento**.

🔹 7.3 Estructura de `addEventListener`

📌 Sintaxis general

```
elemento.addEventListener("evento", funcion);
```

🔍 Elementos:

- `elemento` → nodo HTML
- `"evento"` → tipo de evento (sin `on`)
- `funcion` → función que se ejecuta

⚠️ Error típico

```
// ❌ MAL
boton.addEventListener("click", saludar());

// ✅ BIEN
boton.addEventListener("click", saludar);
```

📌 Se pasa la función, **no se ejecuta**.

🔹 7.4 Comparación directa

Evento en HTML	Evento en JS
`onclick="..."`	`addEventListener`
Fácil al inicio	Profesional
Mezcla HTML y JS	Separación clara
Poco escalable	Escalable
No recomendado	Recomendado

🔹 7.5 Ventajas de usar `addEventListener`

✅ Separación de responsabilidades
✅ HTML limpio
✅ Múltiples eventos en un mismo elemento
✅ Posibilidad de eliminar eventos
✅ Código mantenible

🔹 7.6 Varios eventos sobre un mismo elemento

```
boton.addEventListener("click", function () {
  console.log("Click");
});

boton.addEventListener("mouseover", function () {
  console.log("Ratón encima");
});
```

📌 Con HTML inline esto sería mucho más engorroso.

🔹 7.7 Relación con funciones vistas anteriormente

```
function saludar() {
  window.alert("Hola");
}

boton.addEventListener("click", saludar);
```

📌 Aquí se conectan:

- funciones
- parámetros
- variables
- eventos

🔹 7.8 Idea clave que quiere transmitir la profesora

> **JavaScript debe ser el intermediario entre el HTML y el usuario.**

- HTML → estructura
- JS → comportamiento
- Eventos → conexión entre ambos

8️⃣ Selección de elementos HTML

🔹 8.1 ¿Qué es el DOM?

El **DOM (Document Object Model)** es una **representación en forma de árbol** del documento HTML que el navegador crea al cargar la página.

📌 Gracias al DOM:

- JavaScript puede acceder al HTML
- modificar contenido
- cambiar estilos
- añadir eventos

👉 **El HTML deja de ser estático**.

🔹 8.2 Relación HTML → DOM → JavaScript

```
HTML
 ↓
DOM (árbol de nodos)
 ↓
JavaScript
```

📌 JavaScript **no trabaja directamente con el HTML**, sino con el DOM.

🔹 8.3 Seleccionar un elemento por su identificador (`getElementById`)

📌 Método principal visto en clase

```
document.getElementById("idElemento");
```

🧪 Ejemplo HTML

```
<button id="miboton">Haz click</button>
<script src="app.js"></script>
```

🧪 Ejemplo JavaScript

```
let boton = document.getElementById("miboton");

boton.addEventListener("click", function () {
  window.alert("Botón pulsado");
});
```

🧠 Qué ocurre paso a paso

1. El navegador crea el DOM
2. JavaScript localiza el nodo con ese `id`
3. Guarda el nodo en una variable
4. Se puede trabajar con él (eventos, contenido, etc.)

🔹 8.4 ¿Por qué usar `id`?

📌 El `id`:

- identifica un elemento de forma única
- permite seleccionarlo sin ambigüedades
- es rápido y claro

⚠️ Regla importante:

> En un documento HTML, un `id` **no debe repetirse**.

🔹 8.5 Guardar elementos en variables

Una vez seleccionado el elemento, se suele guardarlo en una variable.

```
let boton = document.getElementById("miboton");
```

📌 Ventajas:

- reutilización
- código más legible
- mejor rendimiento

🔹 8.6 Selección + evento (patrón típico)

Este patrón se repetirá constantemente:

```
let elemento = document.getElementById("idElemento");

elemento.addEventListener("evento", function () {
  // código
});
```

Ejemplo real:

```
let titulo = document.getElementById("titulo");

titulo.addEventListener("click", function () {
  titulo.style.color = "red";
});
```

🔹 8.7 Errores comunes al seleccionar elementos

❌ El elemento no existe:

```
document.getElementById("noExiste"); // null
```

❌ Ejecutar JS antes de que el DOM esté cargado

🔹 8.8 Solución: esperar a que el DOM esté listo

```
document.addEventListener("DOMContentLoaded", function () {
  let boton = document.getElementById("miboton");

  boton.addEventListener("click", function () {
    window.alert("Todo correcto");
  });
});
```

📌 Esto es obligatorio si el JS se carga en el `<head>`.

🔹 8.9 Conexión con el objetivo final

📌 Este paso permite:

- seleccionar elementos
- asociar eventos
- empezar a modificar el HTML desde JS

👉 En la siguiente clase:

- se trabajará con **nodos**
- creación y eliminación de elementos
- DOM dinámico

9️⃣ Eventos del navegador `load` y `DOMContentLoaded`

🔹 9.1 El problema habitual

Situación típica:

```
<head>
  <script src="app.js"></script>
</head>
<body>
  <button id="btn">Botón</button>
</body>
```

```
let boton = document.getElementById("btn");
boton.addEventListener("click", function () {
  alert("Click");
});
```

❌ **Error**:
El botón **aún no existe** cuando el JS se ejecuta.

🔹 9.2 Evento `DOMContentLoaded`

📌 Qué es

El evento `DOMContentLoaded` se dispara cuando:

- el HTML ha sido completamente cargado
- el DOM ya está construido
- **antes** de que carguen imágenes, CSS, etc.

🧪 Uso recomendado

```
document.addEventListener("DOMContentLoaded", function () {
  let boton = document.getElementById("btn");

  boton.addEventListener("click", function () {
    alert("Click correcto");
  });
});
```

📌 Es el evento **más usado** para trabajar con el DOM.

🔹 9.3 Evento `load`

📌 Qué es

El evento `load` se dispara cuando:

- el HTML está cargado
- el DOM está listo
- **todos los recursos** (imágenes, vídeos, CSS) se han cargado

---

🧪 Ejemplo

```
window.addEventListener("load", function () {
  console.log("La página se ha cargado completamente");
});
```

📌 Se ejecuta **más tarde** que `DOMContentLoaded`.

🔹 9.4 Diferencias clave entre `load` y `DOMContentLoaded`

`DOMContentLoaded`	`load`
DOM listo	Página completa
Más rápido	Más lento
Ideal para DOM	Ideal para recursos
Más usado	Uso puntual

🔹 9.5 Cuándo usar cada uno

✅ Usa `DOMContentLoaded` cuando:

- necesitas acceder a elementos HTML

- vas a añadir eventos
- trabajas con el DOM

✅ Usa `load` cuando:

- dependes de imágenes o recursos
- necesitas tamaños reales
- trabajas con multimedia

🔹 9.6 Relación con la práctica de clase

La profesora enlaza el JS en el `<head>`:

```
<head>
  <script src="listener.js"></script>
</head>
```

📌 Esto **obliga** a usar:

```
document.addEventListener("DOMContentLoaded", function () {
  // código seguro aquí
});
```

👉 No es casualidad: es para que entiendas el **orden de carga real**.

🔹 9.7 Alternativa: script al final del `<body>`

```
<body>
  <button id="btn">Botón</button>
  <script src="app.js"></script>
</body>
```

📌 En este caso:

- el DOM ya está creado
- no es obligatorio usar `DOMContentLoaded`
- pero es **menos explícito** a nivel didáctico

🔹 9.8 Error típico de examen

❌ Confundir `load` con `DOMContentLoaded`
❌ Pensar que son equivalentes
❌ Acceder al DOM sin esperar a la carga

🔟 Objetivo final

> Construir una página desde JavaScript y no tener ninguna línea de código en el HTML, solo la carga del fichero JS.

📌 Traducción práctica:

- HTML mínimo (contenedor + `<script src="">`)
- JS crea nodos, añade contenido, gestiona eventos y controla el DOM
